PySnpTools: Reading and Manipulating Genomic Data in Python

Carl M Kadie
Microsoft Research
Redmond, WA 98052
carlk@msn.com

Abstract

PySnpTools is a Python library of reading and manipulating genomic data in Python 3 (and Python 2). It allows users to efficiently select and reorder individuals (rows) and SNP locations (columns). It then reads only the data selected. Originally developed to support FaST-LMM – a genome-wide-association-study (GWAS) tool – PySnpTools now supports large-memory and cluster-scale work.

More generally, PySnpTools was inspired by NumPy and Pandas. It can be thought of as a way to add Pandas-like indexing to 2-D NumPy arrays.

Keywords: Python, Open Source, Genomics, Scalability

Introduction

As we developed the FaST-LMM package (1; 2; 3; 4; 5; 6), we tired of re-writing our code to support more and more file formats (*e.g.*, PLINK's Bed, Dat, Pheno, etc.) (7; 8). Moreover, we noticed ourselves repeatedly performing similar manipulations, for example,

- reading data for just a subset of SNPs (columns),
- reordering the individuals (rows) in our phenotype data to match their order in the SNP data, and
- filling in missing data and normalizing data.

Inspired by NumPy and Pandas, we created PySnpTools, an open-source library that makes these operations easy.

We presented the first public version of PySnpTools at the PyData 2015 conference in Seattle. At the conference, Travis Oliphant (primary creator of NumPy) recommended PySnpTools' approach to reading and manipulating genomic data (9). Not long after the conference, Hilary Finucane (leader of the Finucane Lab at the Broad Institute of MIT and Harvard) wrote us: "I've been loving PySnpTools and recommending it to other statistical geneticists who work in Python (10)!"

Since the conference, as FaST-LMM grew to support datasets of up to 1 million samples (6), we expanded PySnpTools to also simplify:

- larger-than-memory datasets
- running loops on multiple processors or on any clusters, and
- reading and writing files locally or from/to any remote storage.

Most recently, we've added support for Python 3, a much-requested feature.

This paper tells how to install PySnpTools and describes the genomic data that PySnpTools focuses on. It then gives examples of core usage. It closes by listing other PySnpTools features and providing a brief comparison to other dataset approaches.

Installing PySnpTools

To use PySnpTools:

```
pip install pysnptools[1]
```

Find Jupyter notebooks, full API documentation with examples, and source code at https://github.com/fastlmm/PySnpTools.

Genomic Data

The genomic data of interest to us typically consists of 500,000 to 1.5 million columns -- one column for each SNP (that is a genome location where humans are known to differ). The data includes of one row per individual. One thousand to 1,000,000 rows are typical. Values within the initial data might be 0,1,2 or missing (representing the number of minor alleles measured for an individual at a genome location). After standardization, values are 64-bit or 32-bit floats, with missing values represented by NaN ("not-a-number").

Figure 1 shows PySnpTools' in-memory representation of genomic data. Two strings, called the `iid`, identify each individual. One string, called the `sid`, identifies each SNP. A float, called `val`, tells an individual's allele count at a SNP. Finally, a triple of floats, called `pos`, tells the position of each SNP (chromosome number, genetic distance, and base-pair position). For a given position in `val`, PySnpTools makes it easy to find the corresponding `iid`, `sid`, and `pos.` Moreover, for any `iids` or `sids` of interest, PySnpTools makes it easy and efficient to find the corresponding positions in `val`.

For phenotypic and covariate data (*e.g.*, age, sex, height, weight, presence of a disease), PySnpTools uses the same representation. For such data, `sid` tells the name of the feature (*e.g.*, "height") while `pos` is

[1] Assuming pip version 10+ or Anaconda.

ignored. The **val** array, as a float, can also represent binary data via 0.0

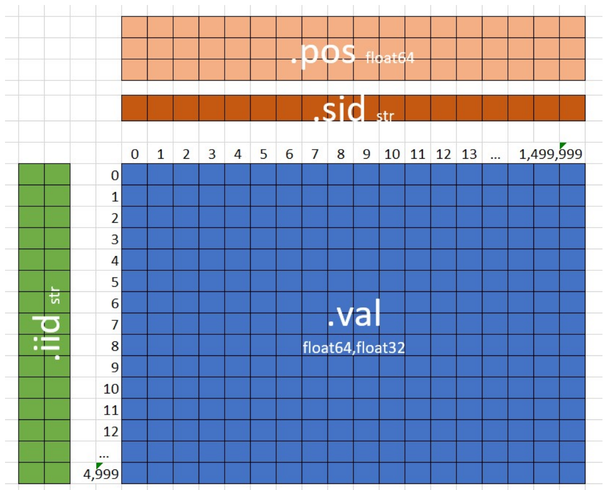

Fig. 1 PySnpTools' in-memory presentation of genomic data

and 1.0. We handle categorical data via hot-one encoding.

Reading Genomic Files: An Example

A Python object that reads genomic data from a file is called a SnpReader. Here is how to create a SnpReader for the popular Bed file format: We tell it what file to read and how to read it. (This SnpReader, as yet, reads nothing from its file.)

Input:
```
from pysnptools.snpreader import Bed
snpreader = Bed("all.bed", count_A1=True)
print(snpreader)
```

Output:
```
Bed('all.bed',count_A1=True)
```

We can ask a SnpReader for the number of individuals and the number of SNPs. We can also, for example, ask it for the **iid** of the first individual. We can additionally ask it for the **sid** of the last SNP. (The Bed SnpReader reads only the small files needed to answer these questions.)

Input:
```
print(snpreader.iid_count,snpreader.sid_count)
print(snpreader.iid[0])
print(snpreader.sid[-1])
```

Output:
```
500 5000
['cid0P0' 'cid0P0']
snp124_m0_.23m1_.08
```

Next, we can read all the genomic data into memory, creating a new SnpReader called a SnpData. Because a SnpData is a SnpReader, we can again ask for the number of individuals and SNPs.

Input:
```
snpdata = snpreader.read()
print(snpdata)
print(snpdata.iid_count, snpdata.sid_count)
```

Output:
```
SnpData(Bed('all.bed',count_A1=True))
500 5000
```

A SnpData is a SnpReader that contains a **val** property that other SnpReaders do not. The **val** property is an (in-memory) NumPy array of the genomic data.

We can, for example, show the genomic data for the first 7 individuals and first 7 SNPs. We can also find the mean of **all** the genomic data.

Input:
```
import numpy as np
print(snpdata.val[:7,:7])
print(np.mean(snpdata.val))
```

Output:
```
 [[0.  0.  1.  2.  0.  1.  2.]
 [0.  0.  1.  1.  0.  0.  2.]
 [0.  0.  1.  2.  1.  0.  0.]
 [0.  0.  0.  2.  0.  0.  0.]
 [0.  0.  0.  0.  0.  0.  2.]
 [0.  0.  1.  0.  0.  0.  2.]
 [0.  0.  2.  1.  0.  1.  2.]]
0.521412
```

Alternatively, we can do everything in one line.

Input:
```
print(np.mean(Bed("all.bed",count_A1=True).read().val))
```

Output:
```
0.521412
```

SnpData

SnpData, the special in-memory SnpReader, is created by any SnpReader's **read()** method. We can also create a SnpData from scratch.

Here we create **snpdata1** for three individuals and two SNPs. We use NaN to mark a missing value and then ask for the mean value (ignoring the missing value).

Input:
```
from pysnptools.snpreader import SnpData
snpdata1 = SnpData(iid=[['f1','c1'],['f1','c2'],
                        ['f2','c1']],
                   sid=['snp1','snp2'],
                   val=[[0,1],[2,.5],[.5,np.nan]])
print(np.nanmean(snpdata1.val))
```

Output:
```
0.8
```

Selecting and Reordering Data *Before* Reading

Suppose we only care about the genomic data for the first 7 individuals and first 7 SNPs. PySnpTools makes it easy to read just the desired data from disk. We use NumPy-like indexing *before* the read method.

Input:
```
snpreader = Bed("all.bed",count_A1=True)
snpdata77 = snpreader[:7,:7].read()
print(snpdata77.val)
```

Output:
```
[[0. 0. 1. 2. 0. 1. 2.]
 [0. 0. 1. 1. 0. 0. 2.]
 [0. 0. 1. 2. 1. 0. 0.]
 [0. 0. 0. 2. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 2.]
 [0. 0. 1. 0. 0. 0. 2.]
 [0. 0. 2. 1. 0. 1. 2.]]
```

All NumPy-like fancy indexing is supported: slicing, Booleans, lists of integers, negative integers (11). Additionally, PySnpTools allows fancy indexing on both rows and columns at once.

Appending indexing to any SnpReader, creates a new SnpReader. Here is an extreme example. It says, "create a reader

- from file "all.bed" in Bed format
- with the individuals in reverse order and for every 2nd SNP
- for the first 5 such individuals and first 5 such SNPs
- for the first and last individuals and the first, 2nd, and 5th SNP,
- then read."

As before, only the final, desired data is read from the disk.

Input:
```
print(Bed("all.bed",count_A1=True)[::-1,::2][:5,:5]
        [[0,-1],[True,True,False,False,True]]
        .read().val)
```

Output:
```
[[0. 0. 1.]
 [0. 1. 1.]]
```

Indexing by Individual (Row) and SNP (Column) Identifiers

We've seen how to manipulate SnpReaders with position-based indexing. What if we instead wish to manipulate SnpReaders by **`iid`** (the individuals' identifier) or **`sid`** (the SNPs' identifier)? This is a more Pandas-like scenario. In that case, we can use the **`iid_to_index()`** or **`sid_to_index()`** methods.

This example shows how to read three SNPs of interest, each identified by **sid** rather than position.

Input:
```
desired_sid_list = ['snp1750_m0_.02m1_.04',
             'snp0_m0_.37m1_.24','snp122_m0_.26m1_.34']
snpreader = Bed("all.bed",count_A1=True)
desired_snpreader = \
    snpreader[:,snpreader.sid_to_index(desired_sid_list)]
print(desired_snpreader.iid_count,
      desired_snpreader.sid_count)
```

Output:
```
500 3
```

Readers and Pseudo-Readers

PySnpTools includes SnpReaders for these common file formats: Bed, Pheno, Dat, Ped, Dense. It also defines these new binary formats: SnpNpz, SnpHdf5, and SnpMemMap. The last is of interest because it uses NumPy's memory-mapped arrays to offer in-memory-like access to data larger than will fit in memory. PySnpTools also defines a format called DistributedBed that offers random access to data stored compactly across dozens (or hundreds or thousands) of Bed files.

In addition, PySnpTools includes SnpGen, which could be called a "pseudo-reader". To any program using SnpGen, it looks like a file reader, but instead of reading data from the disk, it generates random genomic data on the fly. This generation, based on a user-provided seed, is deterministic.

Beyond reading SNPS, PySnpTools includes a set of readers and pseudo-readers for kernel data. (Kernel data represents the pair-wise similarity between individuals.)

If PySnpTools doesn't support a data format of interest, you have two options. If your data is relatively small, you can just read it with other Python tools and then create an in-memory SnpData (or SnpMemMap) object. If more performance is required, you can write a new SnpReader module for the format. In either case, any programs written to expect a SnpReader will automatically work with the new data format.

Beyond Genomic Data

PySnpTools includes PstReader, a generalized class that adds indexing-by-row-or-column-identifier to any 2-D numeric data. It also adds row and column properties. Unlike SnpReader, it does not require rows to be individuals or columns to be SNPs.

Beyond Reading

Recently, as we worked to scale FaST-LMM to 1 million samples, we added more functionality to PySnpTools. PySnpTools now includes tools for:

- standardizing SNP data and kernels,
- intersecting (and ordering) the individuals from any number of SnpReaders (in one line),
- efficiently working with ranges of large integers,
- writing loops so they can run on multiple processor or on any cluster (defined by a module), and
- reading and writing files to any local or any distributed file system (defined by a module).

Comparing to Other Dataset Formats

At PyData 2015, Joshua Bloom compared dataset tools for Data Science (12). Table 1 updates his table with PySnpTools. Fundamentally, PySnpTools decorates 2-D NumPy arrays with immutable, indexed row and column identifiers. This gives us convenience and high efficiency but is more special purpose than other approaches. PySnpTools also delays all copying (and reading) until the **`read()`** method is explicitly called, which gives us efficiency and predictability.

Summary

We have developed PySnpTools, a Python library for reading and manipulating genomic data. Inspired by NumPy and Pandas, PySnpTools can be thought of as a way to add Pandas-like indexing to 2-D NumPy arrays. To install PySnpTools:

```
        pip install pysnptools
```

Jupyter notebooks, full API documentation with examples, and source code are available at https://github.com/fastlmm/PySnpTools.

	Slicing Induces Copy	Immutable Columns	Query	Transfer Speed To Python	C++ SDK	Distributed	Memory Efficiency	Categorical Optimized	Sparse & Dense
PySnpTools	No	Yes (and Row)	Yes	NumPy	NumPy	No	NumPy	No	Dense Only
Pandas DataFrame	Sequences	No	Yes	N/A	No	No	Medium	Medium	Yes
GraphLab SFrame	Yes	Yes	Yes	Low	Yes	Yes	High	No	Yes
Spark DataFrame	Yes	Yes	Yes	Very Low	No	Yes	Low	No	Yes
Dask	Yes	No	Yes	N/A	No	Yes	Medium	No	No
Blaze	No	No	Yes	N/A	No	Yes	Medium	No	No
Wise DataSet	Copy-on-write	No	Yes	Very High	Yes	Yes	Very High	High	Yes

Table 1. Bloom's Dataset for Data Science, plus PySnpTools

Acknowledgments

Thanks to everyone on the FaST-LMM team, whose use and feedback informed the PySnpTools design. A special thanks to Christoph Lippert, who wrote our original Bed reader, and David Heckerman, who encouraged PySnpTools' development.

References

1. *FaST linear mixed models for genome-wide association studies.* **Lippert, C., *et al.*** 2011, Nature Methods, 8 833-835.
2. *An Exhaustive Epistatic SNP Association Analysis on Expanded Wellcome Trust Data.* **Lippert, C., et al.** 2013, Scientific Reports 3, 1099.
3. *Greater power and computational efficiency for kernel-based association testing of sets of genetic variants.* **Lippert, C, *et al.*** 2014, Bioinformatics 30,22.
4. *Further Improvements to Linear Mixed Models for Genome-Wide Association Studies.* **Widmer, C., *et al.*** 2015, Scientific Reports, 4 6874.
5. *Linear mixed model for heritability estimation.* **Heckerman, D. *et al.*** 2016, Proceedings of the National Academy of Sciences 113 (27) .
6. *Ludicrous Speed Linear Mixed Models for Genome-Wide Association Studies.* **Kadie, C. & Heckerma, D.** 2019, bioRxiv 154682.
7. **Purcell, S.** PLINK. [Online] http://zzz.bwh.harvard.edu/plink/.
8. *PLINK: a toolset for whole-genome association and population-based.* **Purcell, S. *et al.*** 2007, American Journal of Human Genetics, 81.
9. **Oliphant, T.** Personal Communications. 2015.
10. **Finuncane, H.** Personal Communications. 2015.
11. **Indexing. *NumPy User Guide.* [Online] https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html.**
12. *A Systems View of Machine Learning.* Bloom, J. Seattle : s.n., 2015. PyData.